

Case-Injection Improves Response Time for a Real-Time Strategy Game

Chris Miles

Evolutionary Computing Systems LAB (ECSL)
Department of Computer Science and Engineering
University of Nevada, Reno
miles@cs.unr.edu

Sushil J. Louis

Evolutionary Computing Systems LAB (ECSL)
Department of Computer Science and Engineering
University of Nevada, Reno
sushil@cs.unr.edu

Abstract- We present a case-injected genetic algorithm player for Strike Ops, a real-time strategy game. Such strategy games are fundamentally resource allocation optimization problems and our previous work showed that genetic algorithms can play such games by solving the underlying resource allocation problem. This paper shows how we can learn to better respond to opponent actions (moves) by using case-injected genetic algorithms. Case-injected genetic algorithms were designed to learn to improve performance in solving sequences of similar problems and thus provide a good fit for responding to opponent actions in Strike Ops which result in a sequence of similar resource allocation problems. Our results show that a case-injected genetic algorithm player learns from previously encountered problems in the sequence to provide better quality solutions in less time for the underlying resource allocation problem thus improving response time by the genetic algorithm player. This improves the responsiveness of the game and the quality of the overall playing experience.

1 Introduction

The computer gaming industry now has more games sales than movie ticket sales and both gaming and entertainment drive research in graphics, modeling and many other areas. Although AI research has in the past been interested in games like checkers and chess [1, 2, 3, 4, 5], popular computer games like Starcraft and Counter-Strike are very different and have not yet received much attention from evolutionary computing researchers. These games are situated in a virtual world, involve both long-term and reactive planning, and provide an immersive, fun experience. At the same time, we can pose many training, planning, and scientific problems as games where player decisions determine the final solution. This paper uses a case-injected genetic algorithm to learn how to play Strike Ops a Real-Time Strategy (RTS) computer game [6].

Developers of computer players (game AI) for popular First Person Shooters (FPS) and RTS games tend to use finite state machines, rule-based systems, or other such knowledge intensive approaches. These approaches work well - at least until a human player learns their habits and weaknesses. The difficulty of the problem means that competent Game AI development requires significant player and developer resources. Development of game AI therefore suffers from the knowledge acquisition bottleneck well known to AI researchers.

Since genetic algorithms are not knowledge intensive and were designed to solve poorly understood problems they seem well suited to RTS games which are fundamentally resource allocation games. However, genetic algorithms tend to be slow, requiring many evaluations of candidate solutions in order to produce satisfactory allocation strategies.

To increase responsiveness and to speed up gameplay, this paper applies a case-injected genetic algorithm that combines genetic algorithms with case-based reasoning to provide player decision support in the context of domains modeled by computer strategy games. In a case-injected genetic algorithm, every member of the genetic algorithm's evolving population is a potential case and can be stored into a case-base. Whenever confronted with a new resource allocation problem (P_i), the case-injected genetic algorithm learns to increase performance (playing faster and better) by periodically injecting appropriate cases from previously attempted similar problems (P_0, P_1, \dots, P_{i-1}) into the genetic algorithm's evolving population on P_i . Case-injected genetic algorithms acquire domain knowledge through game play and our previous work describes how to choose appropriate cases, how to define similarity, and how often to inject chosen cases to maximize performance [6].

Players begin Strike Ops with a starting scenario and the genetic algorithm player attempts to solve P_0 , the underlying resource allocation problem. During game play, an opponent's action changes the scenario creating a new resource allocation problem P_1 and triggering the genetic algorithm to attempt to solve P_1 . Combined with case-injection, the genetic algorithm player uses past experience at solving the problem sequence generated by opponent actions, P_1, P_2, \dots, P_{i-1} , to increase performance on current and subsequent problems, $P_i, P_{i+1}, \dots, P_{i+n}$. Preliminary results show that our case-injected genetic algorithm player indeed learns to play faster and better.

The next section introduces the strike force planning game and case-injected genetic algorithms. We then describe previous work in this area. Section 4 describes the specific strike scenarios used for testing, the evaluation computation, our system's architecture, and our encoding. The subsequent two sections explain our test setup and our results with using case-injected genetic algorithm to increase our performance at playing the game. The last section provides conclusions and directions for future research.

2 Strike Ops

A Strike Ops player wins by destroying opponents' installations while minimizing damage to the player's own installations and assets. An attacking player allocates a collection of strike assets on flying platforms to a set of opponent targets and threats on the ground. The problem is dynamic; weather and other environmental factors affect asset performance, unknown threats can pop up and become new targets to be destroyed. These complications as well as the varying effectiveness of assets on targets make the game interesting and the underlying resource allocation problems difficult and thus suitable for genetic and evolutionary computing approaches. Figure 1 shows a screenshot from the game.

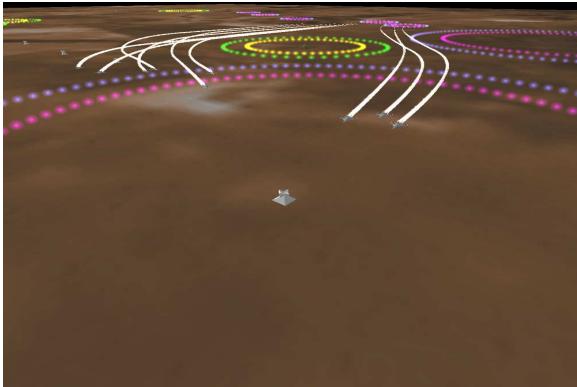


Figure 1: Game Screen-shot

Our game involves two sides: Blue and Red, both seeking to allocate their respective resources to minimize damage received while maximizing the effectiveness of their assets. Blue plays by allocating a set of assets on aircraft (platforms), to attack Red's buildings (targets) and defensive installations (threats). Blue determines which targets to attack, which weapons (assets) to use on them, as well as how to route each platform to the targets, trying to minimize risk presented while maximizing weapon effectiveness.

Red has defensive installations (threats) which protect its targets by attacking enemy platforms that come within range. Red plays by placing these threats to best protect targets. Potential threats and targets can also "pop-up" on Red's command in the middle of a mission, allowing a range of strategic options. By cleverly locating threats Red can feign vulnerability and lure Blue into a deviously located popup trap, or keep Blue from exploiting such a weakness out of fear of a trap. The many possible offensive and defensive strategies make the game exciting and dynamic.

In this paper, a human plays Red while a Genetic Algorithm Player (GAP) plays Blue. The fitness of an individual in GAP's population solving the underlying allocation problem is evaluated by running the game. GAP develops strategies for the attacking strike force, including flight plans and weapon targeting for all available aircraft. When confronted with popups or other changes in situation, GAP responds by replanning with the case-injected genetic algorithm in order to produce a new plan of action that responds to changes.

2.1 Case-Injected Genetic Algorithms for RTS games

The idea behind a case-injected genetic algorithm is that as the genetic algorithm component iterates over a problem it selects members of its population and caches them (in memory) for future storage into a case base [6]. Cases are therefore members of the genetic algorithm's population and represent an encoded candidate strategy for the current scenario. Periodically, the system injects appropriate cases from the case base, containing cases from previous attempts at *other, similar* problems, into the evolving population replacing low fitness population members. When done with the current problem, the system stores the cached population members into the case base for retrieval and use on new problems.

A case-injected genetic algorithm works differently than a typical genetic algorithm. A genetic algorithm randomly initializes its starting population so that it can proceed from an unbiased sample of the search space. We believe that it makes less sense to start a problem solving search attempt from scratch when previous search attempts (on similar problems) may have yielded useful information about the search space. Instead, periodically injecting a genetic algorithm's population with *relevant* solutions or partial solutions to similar previously solved problems can provide information (a search bias) that reduces the time taken to find a quality solution. Our approach borrows ideas from case-based reasoning (CBR) in which old problem and solution information, stored as cases in a case-base, helps solve a new problem [7, 8, 9]. In our system, the data-base, or case-base, of problems and their solutions supplies the genetic problem solver with a long term memory. The system does not require a case-base to start with and can bootstrap itself by learning new cases from the genetic algorithm's attempts at solving a problem.

The case-base does what it is best at – memory organization; the genetic algorithm handles what it is best at – adaptation. The resulting combination takes advantage of both paradigms; the genetic algorithm component delivers robustness and adaptive learning while the case-based component speeds up the system.

The Case-Injected Genetic Algorithm (CIGAR) used in this paper operates on the basis of solution similarity. CIGAR **periodically injects** a small number of *solutions* similar to the current best member of the GA population into the *current* population, replacing the worst members. The GA continues searching with this combined population. The idea is to cycle through the following steps. Let the GA make some progress. Next, find solutions in the case base that are similar to the current best solution in the population and inject these solutions into the population. If injected solutions contain useful cross problem information, the GA's performance will be significantly boosted. Figure 2 shows this situation for CIGAR when it is solving a sequence of problems, $P_i, 0 < i \leq n$, each of which undergoes periodic injection of cases.

We have described one particular implementation of such a system. Other less elitist, approaches for choosing population members to replace are possible, as are different

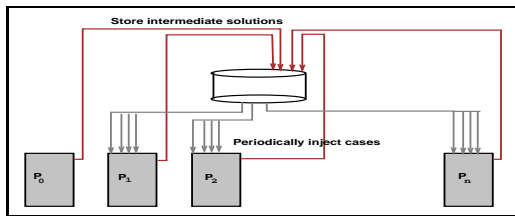


Figure 2: Solving problems in sequence with CIGAR. Note the multiple periodic injections in the population as CIGAR attempts problem P_i , $0 < i \leq n$.

strategies for choosing individuals from the case base. We can also vary the injection percentage; the fraction of the population replaced by chosen injected cases.

Note that CIGAR *periodically* injects cases into the evolving population. We **have to** periodically inject solutions based on the makeup of the current population because we do not know which previously solved problems are similar to the current one. That is, we do not have a problem similarity metric. However, the hamming distance between binary encoded chromosomes provides us a simple and remarkably effective solution similarity metric. By finding and injecting cases in the case-base that are similar (hamming distance) to the current best individual in the population, we are assuming that similar solutions must have come from similar problems and that these similar solutions retrieved from the case base contain useful information to guide genetic search. Results on design, scheduling, and allocation problems show the efficacy of this similarity metric and therefore of CIGAR [6].

An advantage of using solution similarity arises from the string representations typically used by genetic algorithms. A chromosome is, after all, a string of symbols. String similarity metrics are relatively easy to come by, and furthermore, are domain independent. For example, in this paper we use hamming distance between binary encoded chromosomes for the similarity metric.

What happens if our similarity measure is noisy and/or leads to unsuitable retrieved cases? By definition, unsuitable cases will have low fitness and will quickly be eliminated from the GA's population. CIGAR may suffer from a slight performance hit in this situation but will not break or fail – the genetic search component will continue making progress towards a solution. In addition, note that diversity in the population – “the grist for the mill of genetic search [10]” can be supplied by the genetic operators **and** by injection from the case-base. Even if the injected cases are unsuitable, variation is still injected. CIGAR is robust.

The system that we have described injects individuals in the case-base that are deterministically closest, in hamming distance, to the current best individual in the population. We can also choose schemes other than injecting the closest to the best. For example, we have experimented with injecting cases that are the furthest (in the case-base) from the current worst member of the population. Probabilistic versions of both have also proven effective.

Reusing old solutions has been a traditional performance

improvement procedure. This work differs in that 1) we attack a set of tasks, 2) store and reuse **intermediate** candidate solutions, and 3) do not depend on the existence of a problem similarity metric. More details about CIGAR are provided in [6].

Genetic algorithms can be used to robustly search for effective strategies inside our game. A genetic algorithm can generate an initial resource allocation (a plan) to start the game. No plan survives contact with the enemy, however, as the dynamic nature of the game requires re-planning in response to opponent decisions (moves) and the ever changing game-state. Replanning must be done fast enough to keep the game lively and responsive, while plans produced must be effective enough to produce a competent opponent. Can genetic algorithms satisfy these speed and quality constraints?

We have shown that case-injected genetic algorithms learn to increase performance with experience at solving similar problems [6, 11, 12, 13, 14]. For the case-injected genetic algorithm, re-planning is simply solving a similar planning problem arising from opponent actions and this paper shows that when used for re-planning in Strike Ops, a case-injected genetic algorithm quickly produces better new plans in response to changing game dynamics. In our game, aircraft break off to attack newly discovered targets, reroute to avoid new threats, and re-prioritize to deal with changes to the game state.

2.2 Previous Work

Previous work in strike force asset allocation has been done in optimizing the allocation of assets to targets, the majority of it focusing on static pre-mission planning. Griggs [15] formulated a mixed-integer problem (MIP) to allocate platforms and assets for each objective. The MIP is augmented with a decision tree that determines the best plan based upon weather data. Li [16] converts a nonlinear programming formulation into a MIP problem. Yost [17] provides a survey of the work that has been conducted to address the optimization of strike allocation assets. Louis [18] applied case injected genetic algorithms to strike force asset allocation.

From the computer gaming side, a large body of work exists in which evolutionary methods have been applied to games [2, 19, 4, 20, 3]. However the majority of this work has been applied to board, card, and other well defined games. Such games have many differences from popular real time strategy (RTS) games such as Starcraft, Total Annihilation, and Homeworld[21, 22, 23]. Chess, checkers and many others use entities (pieces) that have a limited space of positions (such as on a board) and restricted sets of actions (defined moves). Players in these games also have well defined roles and the domain of knowledge available to each player is well identified. These characteristics make the game state easier to specify and analyze. In contrast, entities in our game exist and interact over time in continuous three dimensional space. Entities are not directly controlled by players but instead sets of parametrized algorithms control them in order to meet goals outlined by players. This adds a level of abstraction not found in more traditional

games. In most such computer games, players have incomplete knowledge of the game state and even the domain of this incomplete knowledge is difficult to determine. John Laird [24, 25, 26] surveys the state of research in using Artificial Intelligence (AI) techniques in interactive computers games. He describes the importance of such research and provides a taxonomy of games. Several military simulations share some of our game’s properties [27, 28, 29], however these attempt to model reality while ours is designed to provide a platform for research in strategic planning, knowledge acquisition and re-use, and to have fun. The next section describes the scenario (mission) being played.

3 Missions

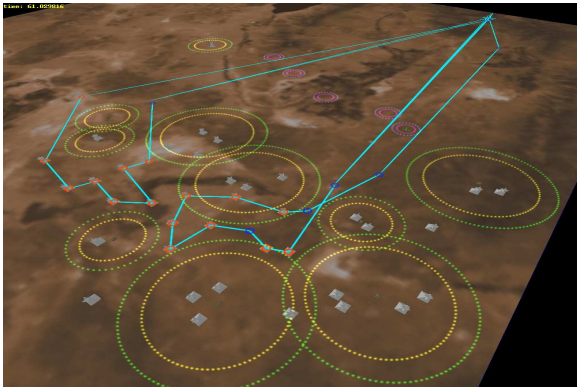


Figure 3: The Scenario

Figure 3 shows one of the missions being played by our GA. This mission takes place in north-west Nevada, Lake Tahoe is just off the lower left hand side of the screen. The rings represent the radii of the various threats, the darker colored rings represent threats that are currently inactive - they are waiting to surprise attackers. The square looking objects are targets and there are a number of them. The blue lines represent routes in a plan designed by a human playing blue. The human player has organized platforms into two wings and plans to thread through gaps in defense coverage to attack undefended targets. Evaluating a plan, or computing a plan’s fitness is dependent on the representation of entities’ states inside the game, and our way of computing fitness and representing this state is described next.

3.1 Fitness

We evaluate the fitness of an individual in GAP’s population by running the game and checking the game outcome. Blue’s goals are to maximize damage done to red targets, while minimizing damage done to its platforms. Shorter simpler routes are also desirable, so we include a penalty in the fitness function based on the total distance traveled. This gives the fitness calculated as shown in Equation 1

$$fit(plan) = Damage(Red) - Damage(Blue) - d * c \quad (1)$$

d is the total distance traveled by Blue’s platforms and c is chosen such that $d * c$ has a 10-20% effect on the fitness

($fit(plan)$). Total damage done is calculated below.

$$Damage(Player) = \sum_{E \in F} E_v * (1 - E_s)$$

E is an entity in the game and F is the set of all forces belonging to that side. E_v is the value of E , while E_s is the probability of survival for entity E . We use probabilistic health metrics to evaluate entity damage.

3.2 Probabilistic Health Metrics

In many games, entities (platforms, threats, and targets in our game) possess hit-points which represents their ability to take damage. Each attack removes a number of hit-points and when reduced to zero hit-points the entity is destroyed and cannot participate further. However in reality, weapons have a more hit or miss effect, destroying entities or leaving them functional. A single attack may be effective while multiple attacks may have no effect. Although more realistic, this introduces a large degree of stochastic error into the game.

To mitigate stochastic errors, we use probabilistic health metrics [30, 31]. Instead of monitoring whether or not an object has been destroyed we monitor the probability of its survival. Being attacked no longer destroys objects and removes them from the game, it just reduces their probability of survival according to Equation 2 below.

$$S(E) = S_{t_0}(E) * (1 - D(E)) \quad (2)$$

E is the entity being considered: a platform, target, or threat. $S(E)$ is the probability of survival of entity E after the attack. $S_{t_0}(E)$ is probability of survival of E up until the attack and $D(E)$ is the probability of that platform being destroyed by the attack and is given by equation 3 below.

$$D(E) = S(A) * E(W) \quad (3)$$

Here, $S(A)$ is the attackers probability of survival up until the time of the attack and $E(W)$ is the effectiveness of the attackers weapon as given in the weapon-entity effectiveness matrix. This method gives us the expected values of survival for all entities in the game within one run of the game, thereby producing a representative evaluation of the value of a plan. As a side effect, we also gain a smoother gradient for the GA to search as well as consistently reproducible evaluations.

3.3 Encoding

To play the game, players get the starting scenario and have some initialization time to prepare strategy. GAP applies the case injected genetic algorithm to the underlying resource allocation and routing problem and chooses the best plan to play against Red. The game then begins. During the game, Red can activate popup threats detectable upon activation by GAP. GAP then runs the case-injected genetic algorithm producing a new plan of action, and so on.

GAP must produce routing data for each of Blue’s platforms. We use the A* algorithm [32] to build routes between locations platforms wish to visit. A* finds the cheapest route, where cost is a function of route length and route risk.

We parameterize the routing algorithm in order to produce routes with specific characteristics and allow GAP to tune this parameter. For example, we have shown that to avoid traps, GAP must be able to specify that it wants to avoid areas of potential danger. In our game, traps are most effective in areas confined by other threats. If we artificially inflate threat radii, threats expand to fill in potential trap corridors and A* produces routes that go around these expanded threats. We thus introduce a parameter, rc that increase threats’ effective radii. Larger rc ’s expand threats and fill in confined areas, smaller rc ’s lead to more direct routes. rc is currently limited to the range $[0, 3]$ and encoded with eight (8) bits at the end of our chromosome. We encoded a single rc for each plan but are investigating the encoding of rc ’s for each section of a route.

Most of the encoding specifies the asset to target allocation with rc encoded at the end as described above. Figure 4 shows how we represent the allocation data as an enumeration of assets to targets. The scenario involves two platforms (P1, P2), each with a pair of assets, attacking four targets. The left box illustrates the allocation of asset A1 on platform P1 to target T3, asset A2 to target T1 and so on. Tabulating the asset to target allocation gives the table in the center. Letting the position denote the asset and reducing the target id to binary then produces a binary string representation for the allocation.

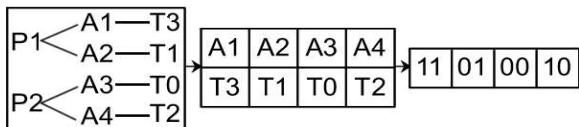


Figure 4: Allocation Encoding

4 Learning from Experience

Previous work has taught us that we can play the game with a GA [30, 31]. Our goal has since become to play the game faster and better. Even on a computer cluster the time required to respond to a change in state is slow compared to the fast pace of the game. We achieve our goal of playing faster and better through case injection.

Case-injected GA’s have been shown to increase performance at similar problems as they gain experience. In strategic games such situations occur often. Every opponent action and situational change is a change in game state. However, these changes are often minor as few opponent actions are worth redeveloping your entire strategy. Case injection maintains some information from previous strategy development, allowing us to keep our current strategy in mind when developing new ones in response to changes in game state. Case injection thus allows the GA to adapt old strategies to new situations, maintaining past knowledge

that is still applicable, and redeveloping new strategies as unforeseen situations arise.

5 Results

In this work we explore how we can improve genetic algorithm performance through case injection. We analyze how case-injection impacts re-planning with respect to game complexity and re-planning scope. Except for the large mission (below), we used a steady-state population size of 20 run for 20 generations with a single-point crossover probability of 0.9 and a mutation probability of 0.1. We injected two individuals every generation chosen using a probabilistic closest to the best strategy where the probability of being picked for injection from the case-base was directly proportional to hamming similarity. The large mission used a population size of 40 run for 40 generations. The algorithm stopped either when it exceeded the set number of iterations (20 or 40) or when there was no improvement for over ten generations or when f_{max}/f_{avg} was less than 1.01. Here f_{max} is the population maximum fitness and f_{avg} is the population average fitness.

5.1 Game Complexity

Game complexity refers to the complexity of the individual mission being played. Increasing the resources available for each side to allocate increases the strategic search space presented to each player. How does this increase in search space alter the effect of case injection on the genetic search?

To test this, we first constructed 3 missions of increasing complexity. More complex missions have more attacking aircraft loaded with more weapons to attack more targets. The mission’s general character does not change. The defending player follows a script activating popup threats early in the mission leading the attacking player (GAP) to replan attacking strategy. Scripting keeps the analysis straightforward and repeatable. We then let the GA play this game multiple times, with and without case injection and analyze the GA’s response. We provide mission profiles below

- Simple Mission
 - 4 platforms
 - 8 assets
 - 8 targets
 - 30 bits per chromosome
- Medium Mission
 - 6 platforms
 - 12 assets
 - 20 targets
 - 66 bits per chromosome
- Large Mission (population size 40)
 - 10 platforms

- 20 assets
- 40 targets
- 114 bits per chromosome

We run a GA with and without case injection on the three missions ten time with different random seeds and plot the average number of evaluations made in Figure 5. Note the **Case-Injected Genetic AlgoRithm (CIGAR)** greatly reduces the number of evaluations (time) to converge outperforming the non-injected GA, especially as the mission becomes more complex. On more complicated missions CIGAR retains more information, giving it a larger advantage over the GA.

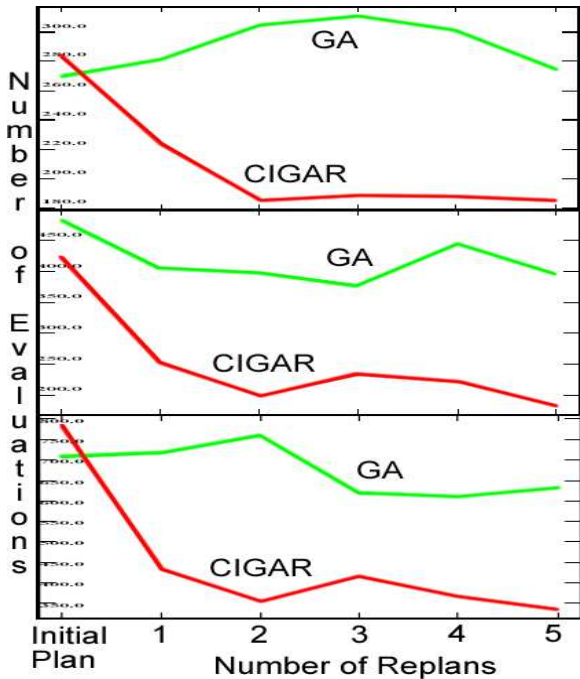


Figure 5: Mission Complexity — Evaluations Made

We can also see that although CIGAR takes less time to converge, the quality of solutions produced by CIGAR does not suffer. Figure 6 plots the average of the maximum fitness found by the GA or CIGAR over ten runs and shows that CIGAR seems to produce better quality solutions; although we need to have more runs to prove the improvement is statistical significant.

5.2 Replanning Scope

GAP re-plans whenever the situation changes. These changes range from minor events like discovering a poorly valued target to big events like highly time-critical and important targets appearing. Case-injection exploits information gained in previous searches, and the scope of the situation change determines how much of that previous knowledge is pertinent to the current situation. How does case-injection work under these different kinds of changes?

We again construct three missions, this time with different defending layouts and scripted actions for the defending player. In each mission the defending player makes five

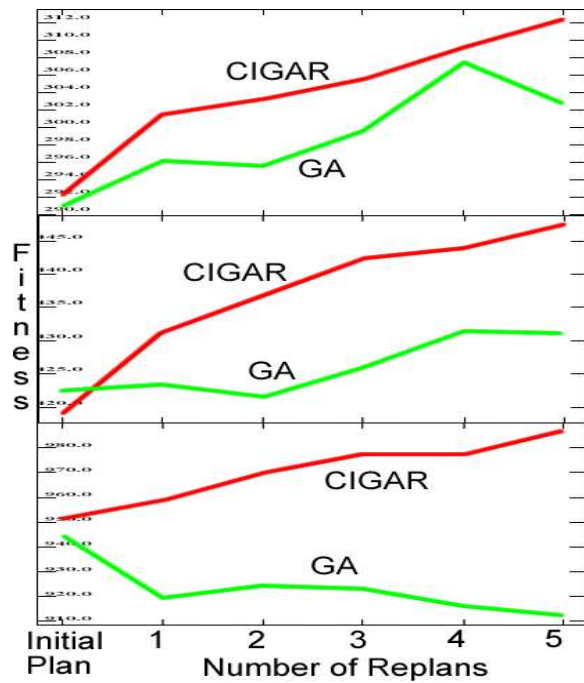


Figure 6: Mission Complexity — Fitness of Solution

changes, these changes having an increasing impact on the attacking player's strategy. We summarize the missions and GAP's response below.

- Simple Replan - Small threats popup on the way to targets (same as previous 3 missions).
 - GA reroutes to avoid new threats
 - Minor changes to weapon-target allocation
- Moderate Replan - Medium threats popup around a handful of targets
 - Moderate changes in allocation
 - Avoids newly protected low value threats
 - Redirects additional attackers to newly protected high value targets
 - Significant routing changes to avoid new threats and reach new targets
- Complex Replan - Large popups occur defending a large cluster of targets
 - Large changes of allocations
 - Wings (groups) of aircraft diverted from new hot zones
 - Focusing of aircraft towards the most highly valued targets
 - Rerouting of most aircraft for each replan

Figure 7 shows the number of evaluations required as a function of the number of re-plans for the above missions. As the scope of the replan increases, case-injection's advantage decreases. In other words, case-injection focuses

search towards previously successful solutions, as the new solution moves further from the old solution the advantage provided by case-injection decreases. Figure 8 shows once again that CIGAR's speed advantage does not come at the expense of lower quality solutions. On the contrary the figure shows that CIGAR produces better quality plans.

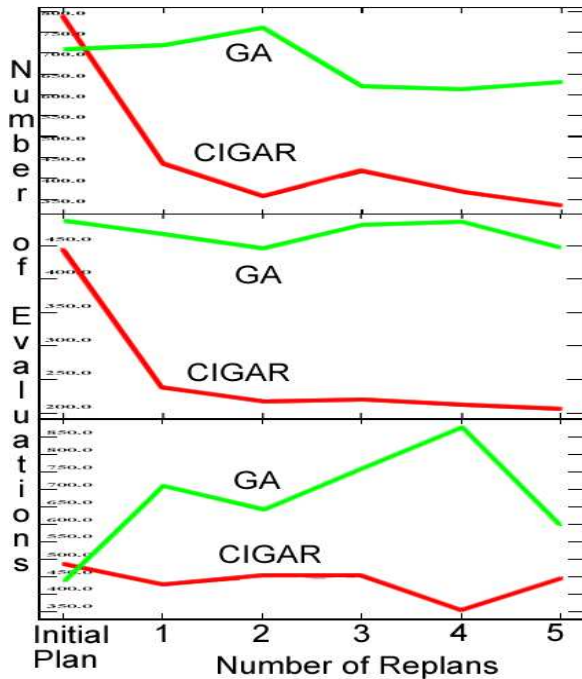


Figure 7: Replanning Size — Evaluations Taken

5.3 Statistical Significance

CIGAR's speedup over the GA is statistically significant. Using CIGAR, we can expect significant speedup with no loss in quality of solution produced. In fact, our results lead us to believe that we should also expect better quality solutions. These results fit well with previous work that shows case-injected genetic algorithms more quickly delivering better quality solutions [6].

6 Conclusions and Future Work

We have shown that a genetic algorithm can play computer strategy games by solving the sequence of underlying resource allocation problems. Case-injection statistically significantly improved the speed with which our case-injected genetic algorithm player (GAP) responds to opponent actions and other changes, while improving the fitness of solutions produced. We explored the effects of mission complexity and replanning scope, showing that the advantage provided by case injection increases as the mission becomes more complicated, and decreases as the difference between new and old situations grows. Note that case injection still provides a significant improvement even when the game situation changes drastically. Playing RTS games with a GA presents a good application of case injection, and we have explored how case-injection impacts the dynamics of the

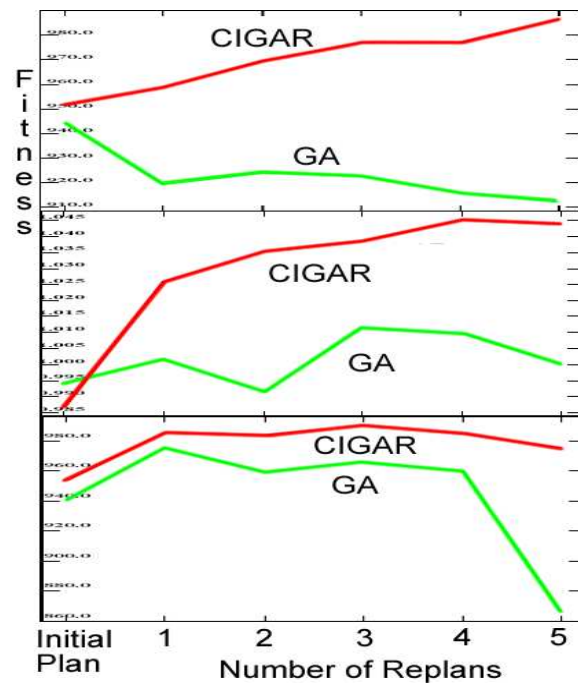


Figure 8: Replanning Size — Fitness of Solution

game, showing significant improvement in response time as well as some improvement in response quality.

We are exploring several avenues for further improving response times. One possibility is to use a reduced surrogate to quickly approximate fitness evaluations. Finally, the game itself is several stages from being human playable and we are currently working on making the game much more playable.

Acknowledgments

This material is based upon work supported by the Office of Naval Research under contract number N00014-03-1-0104.

Bibliography

- [1] Angeline, P.J., Pollack, J.B.: Competitive environments evolve better solutions for complex tasks. In: Proceedings of the 5th International Conference on Genetic Algorithms (GA-93). (1993) 264–270
- [2] Fogel, D.B.: *Blondie24: Playing at the Edge of AI*. Morgan Kaufman (2001)
- [3] Samuel, A.L.: Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development* **3** (1959) 210–229
- [4] Pollack, J.B., Blair, A.D., Land, M.: Coevolution of a backgammon player. In Langton, C.G., Shimohara, K., eds.: *Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA, The MIT Press (1997) 92–98

- [5] Tesauro, G.: Temporal difference learning and td-gammon. *Communications of the ACM* **38** (1995)
- [6] Louis, S.J., McDonnell, J.: Learning with case injected genetic algorithms. *IEEE Transactions on Evolutionary Computation* (To Appear in 2004)
- [7] Riesbeck, C.K., Schank, R.C.: *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Cambridge, MA (1989)
- [8] Schank, R.C.: *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, Cambridge, MA (1982)
- [9] Leake, D.B.: *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI/MIT Press, Menlo Park, CA (1996)
- [10] Goldberg, D.E.: *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley (1989)
- [11] Louis, S.J.: Evolutionary learning from experience. *Journal of Engineering Optimization* (To Appear in 2004)
- [12] Louis, S.J.: Genetic learning for combinational logic design. *Journal of Soft Computing* (To Appear in 2004)
- [13] Louis, S.J.: Learning from experience: Case injected genetic algorithm design of combinational logic circuits. In: *Proceedings of the Fifth International Conference on Adaptive Computing in Design and Manufacturing*, Springer-Verlag (2002) to appear
- [14] Louis, S.J., Johnson, J.: Solving similar problems using genetic algorithms and case-based memory. In: *Proceedings of the Seventh International Conference on Genetic Algorithms*, Morgan Kaufman, San Mateo, CA (1997) 283–290
- [15] Griggs, B.J., Parnell, G.S., Lemkuhl, L.J.: An air mission planning algorithm using decision analysis and mixed integer programming. *Operations Research* **45** (Sep-Oct 1997) 662–676
- [16] Li, V.C.W., Curry, G.L., Boyd, E.A.: Strike force allocation with defender suppression. Technical report, Industrial Engineering Department, Texas A&M University (1997)
- [17] Yost, K.A.: A survey and description of usaf conventional munitions allocation models. Technical report, Office of Aerospace Studies, Kirtland AFB (Feb 1995)
- [18] Louis, S.J., McDonnell, J., Gizzi, N.: Dynamic strike force asset allocation using genetic algorithms and case-based reasoning. In: *Proceedings of the Sixth Conference on Systemics, Cybernetics, and Informatics*. Orlando. (2002) 855–861
- [19] Rosin, C.D., Belew, R.K.: Methods for competitive co-evolution: Finding opponents worth beating. In Eshelman, L., ed.: *Proceedings of the Sixth International Conference on Genetic Algorithms*, San Francisco, CA, Morgan Kaufmann (1995) 373–380
- [20] Kendall, G., Willdig, M.: An investigation of an adaptive poker player. In: *Australian Joint Conference on Artificial Intelligence*. (2001) 189–200
- [21] Blizzard: Starcraft (1998, www.blizzard.com/starcraft)
- [22] Cavedog: Total annihilation (1997, www.cavedog.com/totala)
- [23] Inc., R.E.: Homeworld (1999, homeworld.sierra.com/hw)
- [24] Laird, J.E.: Research in human-level ai using computer games. *Communications of the ACM* **45** (2002) 32–35
- [25] Laird, J.E., van Lent, M.: The role of ai in computer game genres (2000)
- [26] Laird, J.E., van Lent, M.: Human-level ai's killer application: Interactive computer games (2000)
- [27] Tidhar, G., Heinze, C., Selvestrel, M.C.: Flying together: Modelling air mission teams. *Applied Intelligence* **8** (1998) 195–218
- [28] Serena, G.M.: The challenge of whole air mission modeling (1995)
- [29] McIlroy, D., Heinze, C.: Air combat tactics implementation in the smart whole air mission model. In: *Proceedings of the First International SimTecT Conference*, Melbourne, Australia, 1996. (1996)
- [30] Miles, C., Louis, S.J., Drewes, R.: Trap avoidance in strategic computer game playing with case injected genetic algorithms. In: *Proceedings of the 2004 Genetic and Evolutionary Computing Conference (GECCO 2004)*, Seattle, WA. (2004) 1365–1376
- [31] Miles, C., Louis, S.J., Cole, N., McDonnell, J.: Learning to play like a human: Case injected genetic algorithms for strategic computer gaming. In: *Proceedings of the International Congress on Evolutionary Computation*, Portland, Oregon, IEEE Press (2004) 1441–1448
- [32] Stout, B.: The basics of a* for path planning. In: *Game Programming Gems*, Charles River media (2000) 254–262